Phase 2 Report

Overall approach to game implementation

Development of this game began first from a creative perspective. Phase 1 let us explore ideas that we thought would be interesting to implement in our 2D game, and we settled on the idea of an SFU student escaping campus. From the start of implementing this game, we began looking at the types of process models that would best fit the needs of our project. After phase 1 clearly outlined the requirements of the game, we thought about the different aspects that would lead to the decision we made about the process model to pursue. Aspects considered included: uncertain design requirements, only four developers, which means each member has a high amount of responsibility and accountability; and unpredictable working hours, which could lead to organizational turmoil. Due to these reasons, we decided to apply an agile approach to our software development. This meant we agreed on design requirements early on in the process, knowing that we would likely adjust multiple times later on.

Our division of roles and responsibilities broadly reflected our software development approach. We split much of the workload on overseeing different parts of the project. In our own sections, we would have the most responsibility with progressing development and making foundational decisions. We had one member oversee most of the game logic and game-related implementation, one member in charge of the menu screens, one in charge of the artistic development, and one who focused on journaling our progression. Most of the workload came with the game itself, so everyone assisted when they could.

With our development approach chosen, the design of the game began with a set of principles we wanted to follow, these included functionality, usability, reliability, performance, and supportability (FURPS). Functionality is important as the game needs to work as expected, but usability, reliability, performance, and supportability was vital to us in ensuring the game was clear in its instructions, didn't crash even in unique situations, ensuring the game was predictable and felt fluid, and making sure we can easily make changes to the game in case we ever need to. This was done in our game in a variety of different ways, but chief among them was abstraction. Abstraction was used for creating every object of the game, including (but not limited to) the main character, enemies, rewards, and even the maps themselves. Most, if not all, classes that were created could be extended to include a different type of that object if we wanted to add more to the game. Adding something like a new type of reward, or a final boss enemy is entirely possible with this implementation.

Starting out with developing a project of this size felt overwhelming initially, but subdividing this complex problem made progression easier. Modularity was utilized to divide the project into named addressable components. In the context of our game, we separated the game into components such as the game's engine, all the characters, rewards, collision detection, input from the user, maps, the appearance of the game, and more. Separation into these components can be seen in our directory layout and allowed for easier planning, distribution of tasks among the group, easier implementation of changes in a team setting, and made for more efficient testing and debugging as each component was tested individually prior to integration.

The next step in implementation was looking into design patterns. We wanted to avoid depending on specific operations too much, and we wanted to avoid creating objects by specifying a class directly. We thought this was necessary because we believed many aspects of our game would use similar methods (i.e., a static enemy vs. a moving enemy). Of the three design patterns we were familiar with, we felt that creational patterns made the most sense for this project given that we wanted to, primarily, abstract the process of creating objects. The flexibility that this design pattern provides in terms of how objects are created and the types of objects we are able to create made this design pattern the clear choice for our needs. The use of abstract factories to accommodate the different types of objects we had, as well as further the specific types of each object (i.e., a bonus reward extends a reward) resulted in a less hard-wired and more robust game.

Our game relies on a thread that begins when the game is started by the user. We implemented this thread using method 1 as discussed in class (by providing a Runnable object). This thread continues until the game is ended by the user. The use of a thread is integral to the functioning of most games as it allows tasks to occur in the background while the user is uninterrupted while they play the game. In terms of the user interface design of our game, we began with paper prototyping of what we expected our game to look and feel like. We expected the average user to know how to operate the basics of a keyboard when making our UI design. To ensure our game was as usable as possible, many of Nielsen's 10 principles were implemented while UI design was done. Principles that were implemented include matching the real world (visual feedback and immediately visible effects such as moving the character and it faces the way you pressed it, or the door indicates that it is the exit for a map), help and documentation (how to play guide), recognition (symbols are used for rewards and penalties throughout the game), and error reporting/recovery. We also included the aspect of minimalist design in that most designs made for our game were done with simplicity as a goal. We aimed for a traditional video game aesthetic that did not overly rely on colour differences for the user to differentiate where the main character could and could not access, which is why differently textured areas were included in the maps. In addition, aspects of exception handling were implemented as we had numerous instances of try and catch, primarily when loading in character images for our design, but also when we used methods such as getMessage() and printStackTrace().

Adjustments and modifications made to the initial design of the project

The initial UML design, made prior to our implementation, was scrapped as our initial design of a board class hierarchy did not make much sense in terms of the inheritance between our different classes and the needs of our game. Instead, we created several different main classes and had their own respective methods. In our case, we created separate main classes such as for character, reward, and abstract factories. This made more sense for our implementation because we were able to separate all the various aspects of the game easily

When it came to the UI of the game, the blueprint we planned was generally followed. That said, almost all of the smaller specifics were altered, or even abandoned. This can be seen when it comes to the menu screens. For the most part, the current design of the screens are consistent with the simple and unpopulated mockups drawn previously. However, the layouts for all of the screens, excluding the 'How to Play' screen, have been altered. This was mostly due to the unforeseen challenges related to utilizing the different layouts available. Originally, we thought we could implement the screens using the BorderLayout from javax.swing.* such that they would closely resemble the mockups drawn. It became apparent fairly early that this was not the case. In an attempt to achieve near resemblance to the mockups, we even tried to implement layouts from the JavaFX library. In the end, we decided that it was not critical to the quality of the program. We ended up utilizing the FlowLayout from javax.swing.* as it was especially capable at achieving the uncomplicated and clear layout that was important to us. Additionally, we decided against a few features, such as a settings screen, a choose difficulty screen, and an exit confirmation screen. These were mostly as a result of the changing construction of the game itself. Furthermore, the heads-up display of the game also completely changed. Unlike the menu screens, this was due to concluding that the mockups drawn were not optimal.

In hindsight, if we were to carry out a similar project, we would spend more time thinking about, and researching, the possible challenges we would face during the software development cycle as it would lead to a few complications when it came to putting everything togeather.

List of libraries used

Library Used	**Reason for Use**
javax.swing.JPanel;	Adding components to and customizing panels, setting screen size, background colour, double buffering
java.awt.Dimension	Help set size of the screen on which game is displayed
java.awt.Color	Set background colour of game screen
java.awt.Font	Used to create and modify font objects
java.awt.Graphics/Graphics2D	Paint components in the game every frame
javax.imageio.ImageIO	Load images into methods (such as enemies into getMovingEnemySprite())
java.awt.Random and java.util.Random	Generate random numbers for the attributes of some of the characters
java.awt.BufferedImage	Handle and manipulate custom images used to create designs for characters, levels, objects
java.io.IOException;	Catch the scenario where images could not be loaded in the getPlayerSprite() method
java.io.InputStream, java.io.InputStreamReader and java.io.BufferedReader	To read read file of game's map information
java.awt.event.KeyEvent and java.awt.event.KeyListener	Detect when a key is pressed or released in relation to player movement, and when there is a change of state
import javax.swing.*;	This was used for the GUI of the menu screens. FlowLayout and BorderLayout seemed to be the most adequate for the simple aesthetics of the mocked-up designs menu screens. Also, the swing

	library seemed easier to handle design changes if it came to it compared to more complex alternatives, such as JavaFX library. Ultimately, the FlowLayout was chosen due to its linear design.
javax.swing.border.EmptyBorder;	Used to add empty borders for the menu screens. They help maintain the structure of the screens through the window size changing.
java.awt.*	Used to manage the graphical user interface of the menu screens (such as with Dimension).
java.awt.event.*	Used in the menu screens for ActionListener interface and to respond to button interactions with the user.
javax.swing.plaf.metal.MetalButtonUI	This class from the javax.swing package was used to make all of the buttons on the menu screens have metal aesthetics. This UI has qualities that resemble the SFU logo.